PATENT
260/087

S P E C I F I C A T I O N

SYSTEM AND METHOD FOR SOFTWARE CODE OPTIMIZATION

This application claims priority to a U.S. Provisional Application entitled "System-on-a-Chip-1," having Serial No. 60/216,746 and filed on July 3, 2000, and which is hereby incorporated by reference into this application as though fully set forth herein.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to the field of software development, and particularly, to system and methods for software code optimization.

2. Background

In the design of software for digital signal processing (DSP) and other applications, programmers take advantage of the low-level but high-speed capabilities that the particular target processor (e.g., DSP processor, microcontroller) offers in order to achieve the performance requirements for the applications. However, the application of these tools early in the development process leads to the development of a program that may be unportable should a different target processor be subsequently used to host the program. The development of code that is not portable from one target processor to another may result in significant

redesign and development costs for the same basic application. Often, if the program is portable, the most significant issue is the cost in time and resources to port the application to a new host, such that the performance requirements of the program on the new host are met.

A need exists therefore for a system and method that minimizes the likelihood that a development process for software will result in a program that is unportable from one target processor to another.

SUMMARY OF THE INVENTION

The present invention, in one aspect, provides a systems and methods for optimizing software for execution on a specific host processor.

In one embodiment, a method is provided of optimizing a software program for a target processor in order to meet specific performance objectives, where the software program is coded in a high-level language. The method includes the steps of first optimizing the software program in the high-level language, using optimizations that are substantially independent of the target processor to host the application. Preferably, if the performance objectives are met after the completion of this step, then the process preferably successfully exits. Thus, if the performance objectives are not met, then the method preferably proceeds to a second step.

In the second step, the initially optimized form of the software program is again optimized in the high-level language, although target processor-dependent

optimizations are used. If the performance objectives are met after completing this second step, then the process preferably terminates. If the performance objectives are not met, then the process proceeds to a third step.

In the third step, the twice-optimized software program is optimized using a low-level language of the target processor on key portions of the code, such that although the software implementation becomes target-dependent, it remains relatively portable. Preferably, in evaluating whether the performance objectives have been achieved, performance profiles are determined for the intermediate forms of the optimized software program. These performance profiles are then preferably quantitatively compared to the previously defined performance objectives.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a flow diagram generally depicting steps in a process of optimizing a software program for a target processor representing a preferred embodiment of the invention.

FIG. 2 is a flow diagram depicting a preferred embodiment of a simplified representation of the main steps of the generic implementation process represented as one step in FIG. 1.

FIG. 3 is a flow diagram depicting a preferred embodiment of detailed steps of the generic implementation process depicted in FIG. 2.

FIG. 4 is a graph depicting examples of curves of the evolution of a software application with respect to its performance and size in a target-independent optimization process.

FIG. 5 is a flow diagram depicting a preferred embodiment of a simplified representation of the main steps of the specific implementation process represented as one step in FIG. 1.

FIG. 6 is a flow diagram depicting a preferred embodiment of detailed steps of the specific implementation process depicted in FIG. 5.

FIG. 7 is a graph depicting examples of curves of the evolution of a software application with respect to its performance and size in a target-dependent optimization process.

FIG. 8 is a flow diagram depicting a preferred embodiment of steps of the fully dedicated implementation process represented as one step in FIG. 1.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

In a preferred embodiment of a software code optimization method (or process) comprising multiple basic steps, each successive basic step generally results in code that is closer to being dedicated to operate on a particular target. Thus, to promote portability, if the performance goals of the application are reached

after the completion of any step in the process, then the optimization process is terminated.

The evaluation of performance, and thereby, whether the performance meets stated, predefined objectives preferably accounts for several factors. Preferably, a key performance measure is the real-time speed of the application when operating on the specified target. Another performance measure is the accuracy and/or quality of the output. Another factor that may be integrated into the process evaluation is the binary code size. While the application is made to fit in the target processor's memory, this factor generally is less and less important as memory sizes increase, become smaller, cheaper and less power consuming.

Thus, one major step in the optimization process is to fix the initial constraints that are applied to the development and optimization of the software application. These constraints are preferably used to quantitatively evaluate the application implementation's performance in an overall sense, and facilitate determining the feasibility of porting the application to a specific target at each of the development stages. These measures preferably inherently integrate processing performance characteristics of the target processor, including its clock frequency, which relates to the number cycles available to execute the application.

Another set of parameters that preferably are calculated is the global I/O data flow to determine if the memory accesses (read/write) are achievable for the

specified target. This set of parameters integrate elements like the aggregated data flow over the internal and external buses (data exchanges).

FIG. 1 is a flow diagram generally depicting steps in a process of optimizing a software program for a target processor. In the embodiment represented in FIG. 1, an optimization process 100 comprises three optimization steps. In a first optimization step 102, the software for the DSP processor target is written in a high-level language such as C, C++ or Ada. Preferably, the programming language is one that is completely portable between all probable DSP targets. In coding the software, optimization techniques particular to the language are preferably used. The code optimization in this step 102 preferably does not employ any optimization tools that depend on the processor that is meant to host the application.

Upon completion of this step 102, the new implementation, as a next step 104, is evaluated to determine whether the performance goals have been reached. If by completing the step 102 of target-independent optimization the performance objectives are achieved, then the overall optimization process 100 successfully terminates 112. If the performance requirements for the application have not been achieved, then in a next optimization step 106, certain portions of the software code are re-implemented in the high-level language to take advantage of the specific processing capabilities of the DSP target.

While the code after this step 106 is less portable than the code that results after the previous step 102, the software may remain partially portable for a number of reasons. One reason is that the modified code is preferably selected from a portion of code that is short in terms of lines of source code, but is repeatedly executed and is thus responsible for a relatively significant percentage of the processing overhead. By first modifying the code that fits these criteria, the amount of code that must be modified is minimized, and may additionally be flagged in the source file to indicate that it is target–specific code. If the target processor later changes, only these identified portions need be addressed for optimization. Further, the previously unoptimized code, corresponding in functionality to the portion of code that is optimized in this step 106, may remain coded in the source file. This original unoptimized code may be used as a starting point for optimizing the same portion of code of any subsequent target processor. Another benefit is that although the coding is specific to the DSP target for the application, the code preferably remains in the high-level language. By remaining in a high-level language (versus being re-coded in a low-level language such as an assembly language), the resulting code is inherently much easier to revisit and comprehend should modifications be necessary.

Preferably, software-profiling tools are applied to readily identify the portions of code that fit the criteria required to be preferred candidates for optimization, so that they then can be optimized for the particular DSP target as necessary.

Once a section of code has been optimized for a particular DSP target, other portions of code that meet the criteria to be candidates for optimization may also be optimized for the DSP target, if the performance criteria for the application have still not been achieved.

If the portions of code that are candidates for optimization have been optimized, then in a next step 108, the implementation is again evaluated to determine whether the performance objectives on the target processor have been met. As in step 104, if the performance objectives are achieved though the step 102 of target-dependent optimization using the high-level language, then the overall optimization process 100 successfully terminates 112. However, if the performance goals of the application have not been met, then the optimization process 100 proceeds to a third optimization step 110. In this step 110, the software is configured to be fully dedicated to the architecture and processing benefits of the target processor. Various coding techniques that are particular to the target processor for the application may be employed. Some of these techniques include executing instructions in parallel or using any pipeline processing or other specialized processing capabilities. Further, tradeoffs may be made between performance and throughput in order to meet pre-stated objectives of the application. The result of the process is an efficiently created program for a DSP, microcontroller, or other computing target processor that meets pre-stated performance objectives, and is optimal, or close to optimal, with respect to its

portability, thereby minimizing future software development efforts for the same application.

In one embodiment of a system for performing the optimization method, the method is performed automatically after the software code has been initially developed in a high-level language. Preferably, the system is provided the performance parameters that are desired for the application, as well as the architectural specification of the target processor. Given these inputs, the system then processes the high-level language source code, compiles and simulates the code's execution, and tests the code against the specified performance requirements. If the performance requirements are not met, the system profiles the code and then optimizes the portions that are the best candidates for optimization.

Preferably, the system comprises a software-optimizing processor in conjunction with memory that automatically performs the code profiling operations, code generation operating on portions of code that are determined to be candidates for optimization, and then subsequent performance analysis. The software-optimizing processor may comprise any type of computer, and has processing characteristics dependent upon, for example, the processing requirements for the code generation, profiling and performance assessment operations. It may comprise, e.g., a computer, such as a workstation such as are manufactured by Sun Microsystems, a main frame computer, or a personal computer such as the type

manufactured by IBM or Apple. A computer executing optimization software is preferably used for the software-optimizing processor, due to the utility and flexibility of a computer in programming, modifying software, and observing software performance. More generally, the software-optimizing processor may be implemented using any type of processor or processors that may perform the code optimization process as described herein.

Thus, the term "processor," in its use herein, refers to a wide variety of computational devices or means including, for example, using multiple processors that perform different processing tasks or have the same tasks distributed between processors. The processor(s) may be general purpose CPUs or special purpose processors such as are often conventionally used in digital signal processing systems. Further, multiple processors may be implemented in a server-client or other network configuration, as a pipeline array of processors, etc. Some or all of the processing is alternatively implemented with hard-wired circuitry such as an application-specific integrated circuit (ASIC), a field programmable gate array (FPGA) or other logic device. In conjunction with the term "processor," the term "memory" refers to any storage medium that is accessible to a processor that meets the memory storage needs for a system for optimizing software. Preferably, the memory buffer is random access memory (RAM) that is directly accessed by the software-optimizing processor for ease in manipulating and processing selected

portions of data. Preferably, the memory store comprises a hard disk or other non-volatile memory device or component.

Preferred embodiments of the method for performing each of the basic steps illustrated in FIG. 1 are now provided.

Generic Implementation Process (Target independent Optimization)

As used herein, the term generic means target-independent. In the DSP domain, in a target-independent implementation, the high-level language source code, normally C- code, uses no specific function calls to pragma or macros dedicated to the target. With a target-independent implementation, the portability of the application is maintained and some optimization is integrated into the application at a high level, without using assembly language code.

FIG. 2 is a flow diagram depicting a simplified representation of the main steps of the generic implementation process 200 represented as step 102 in FIG. 1. Preferred detailed steps of this process 200 are depicted in the task flow diagram of FIG. 3. Preferably, as shown in FIG. 2, there are four main steps that take as input the mathematical theory related to a signal processing algorithm and lead to an implementation that is later used by the specific implementation process.

Stage G1: Floating Point Implementation

The floating-point implementation step takes as input the theoretical solution of a process and transforms the solution into a structured language implementation.

A main purpose of the step is to be able to reflect as much of the math in the theory into the implementation.

Precision in the calculations is important in the floating-point implementation. In general cases, those applications are done using double-floats and tools like the Cadence® Cierto™ signal processing worksystem or MathLab. Such tools provide representation of the processed data, allow graphical representation and comparison, and extract errors so that an implementation can be qualified.

For an integer DSP, the floating-point implementation transitions to a fixed-point implementation linked to the precision that the DSP can handle. For example, the DSP may need a 16-bit precision implementation. However, typically a group developing the floating-point application is not the group developing the fixed-point implementation. This means that there are at least the following two approaches. In the first approach, the theoretical implementation is made with no consideration of the precision. In that case, the implementation is oriented to processing quality and pushes the precision problem to the fixed-point porting. In the second approach, a target precision is involved at an early stage of the development and impacts the quality of the processing. This provides a full precision-oriented implementation. However, this implementation must be entirely redone if the target architecture is changed. Details regarding floating point formats

and related issues in terms of implementation are provided in several articles, including Morgan, Don, "Practical DSP Modeling, Techniques, and Programming in C," John Wiley & Sons, 1995, pp. 263-298, and Lasley, P., Bier, J., Sholan, A., and Lee, Edward A., "DSP Processors Fundamentals, Architectures and Features," IEEE Press Series on Signal Processing, 1997, pp. 1-30, which are hereby incorporated by reference as though fully set forth herein.

Stage G2: Fixed Point Implementation

In the stage of deriving the fixed-point implementation, trade-offs relating to precision may be made. The extent of these trade-offs primarily depends on the target DSP capability. If the target processor is a 16-bit precision DSP, the accepted deviation of the output result will be greater than on a 32-bit DSP.

However, depending on the complexity of the algorithm, another factor, the implementation architecture, is preferably considered. If an implementation involves hundreds of function calls, the real-time execution at the end of the implementation flow is impacted. For this reason, two different steps in the implementation activity are utilized.

Another consideration at this level is the inheritance. A common method of implementing signal processing is to take the floating-point implementation and port it to a specific target. Another method includes porting an existing fixed-point implementation to a new target. The mechanisms are quite different because of the

availability of a first implementation. In the latter case, it is more an adaptation of an existing application than a new implementation. The advantage is that it shortens the development process by reusing the existing code done for another target.

Sub-stage G2.1: Processing Qualification

The goal of processing qualification is to obtain an implementation that preferably provides the best trade-off on the output result for a given precision. One of the tools that can accelerate the completion of this step is the Cierto™ signal processing worksystem. This tool provides the capability to validate, compare, and qualify a process with a reference to the floating-point implementation.

Fixing the derivation criteria depends primarily on the application category. For image processing comparison, information like texture, edge, contrast and distortion is considered. For voice processing, the same elements may be taken into account, but spectral analysis, tone, volume and saturation, etc. may also be considered. Depending on the application domain, the criteria can be completely different. Furthermore, within a given domain, the criteria can change. Radar can be used in military or agricultural activities but the measures made for those two applications of radar image may be quite different.

Sub-stage G2.2: Implementation Sizing

With a qualified algorithm in terms of quality and precision, the first sizing of the algorithm can be addressed. Preferably, the information gathered includes the real-time data flow, the implementation structure and architecture, the profiling of instructions and cycles, and the performances of the target DSP. These elements help determine if the code can fit inside the target.

Real-Time Data Flow

The goal of real-time data flow is to understand the different I/Os related to the algorithm that are to be integrated into the DSP. On one level is the global data flow that globally indicates the availability of the raw and the processed data. With the global data flow, the developer identifies the processing delays that are going to provide a basic characteristic of the application relating to data flow.

However, with the global data flow that corresponds to a simplified representation of the data flow, the real behavior of the data coming in and out on the data bus of the system is not necessarily clear. The programmer may have to zoom in the elementary time duration (selected for the global data flow representation) to characterize the behavior of the implementation confronted with the interruptions coming from the devices involved in the process. This "elementary time duration" can be very different from one application to another. It can be the duration of, for example, an image frame, an image line, an audio frame, or a dedicated time dictated by control software or the processor.

Another data flow consideration is application cadence. Application cadence may impact all future decisions for the application. For example, in an interrupt-driven architecture, which is the case in most of the real-time DSP constraint developments, it is then possible to make clear design choices like use of a (first-in first-out) FIFO that will buffer data. This option provides a more flexible way to manage bus I/Os because it allows a better optimization of the bandwidth usage. It is generally a more expensive system design, but it is recommended for processing that involves large amounts of data, like image processing.

Alternatively, a designer may choose not to use a FIFO. This means that each piece of data produced is either immediately saved or eventually lost. This is the most constrained way of implementing a signal processing application, but is cheaper and well suited for processing that involves little data such as voice processing. This example shows the impact of the application cadence criteria on application development.

Another factor is bandwidth. Such a study exhaustively integrates external data variables and code fetches. However, a strict and exact representation of the detailed I/Os is generally impossible. All of the representations reflect only the static point of view. A real I/O study preferably indicates that temporal drift impacts the overall bandwidth all along the application processing.

Implementation Organization

Also affecting implementation sizing is implementation organization. Implementation organization preferably considers the implementation structure, the architecture of the implementation, and the behavior of the implementation.

Implementation Structure

The implementation structure generally means that the developer knows the number of functions implemented, the number of times they are called, split if possible into low-level and high-level functions, and so on. This first measure can be made manually or by using tools. One difficulty is identifying a tool that indicates the number of times a function is called. Context switching can be expensive if it occurs too many times. For this purpose, one can use free coverage tools like *gprofs* that provide part of the necessary information. The use of other tools like Sparcworks (Sun Microsystems) provides *the call graph*.

Implementation Architecture

The architecture of the implementation generally means knowing the overall behavior of the application to know if it may be necessary to revisit the algorithm construction to emphasize real-time issues. Given a specific processing algorithm that produces a signal processing development, the requirements can be formalized as follows:

1) Obtain an "elementary" signal sample. This can be an audio value, an entire image, etc.

2) Process the sample using the development made.

The first step in evaluating the feasibility of the application includes determining the global data flow to fix the limit of the input/output size and the precision (8, 16 or 32 bits). The first output of the data flow indicates if it is possible to sustain the I/Os, but another indication concerns the algorithm structure.

Effectively analyzing the processing flow in conjunction with the data flow can indicate that the some steps of the processing cannot be done before others. In that case, it may be possible to conclude sometime that some delay constraints can not be matched or simply that the algorithm cannot run real-time.

There are several definitions of the delay. There is the intrinsic delay related to every processing called real processing delay. In a data process, there is always the processing delay needed to perform the data transformation. But there is also the Architectural Delay (AD) related to the structure of the algorithm. In this case, it is related to the algorithm architecture that never allows reducing the architectural delay.

Application Behavior

A majority of applications integrate some computations that use static correspondence tables or lookup tables to transform the signal. However, depending on the calculation results, the tables that are used will not be the same.

If, for the same computation, the signal processing uses two different conversion tables that have different sizes, then the application is non-deterministic. Thus, this part of an implementation is preferably clearly identified so that all of the steps that follow result in useful measures that can be accurately correlated with the performance increase of the application.

Profiling

The objective of high-level profiling is to provide a first indication of the number of cycles consumed by the implementation and the binary code size. If necessary, a simulator can also produce an instruction profiling.

One difficulty is fixing the comparison criteria so that it is known whether the application fits in the targeted DSP. However, it is possible to get a ratio based on the different benches realized on the DSP. The benches are generally provided by the DSP vendors. This means that the cycle counts indicated at a higher level must be correlated with the performances of the target DSP to establish a go/no go process.

As an example, several DSP providers provide appropriate benches. Several DSPs are compared with C-written kernel functions including MAC: Multiply accumulate, Vect: Simple vectors multiply, Fir: FIR filter with redundant load elimination, Lat: Lattice synthesis, Iir: IIR filter, Jpeg: JPEG discrete cosine transform

From the application point of view, if the programmer takes the average cycle reduction ratio, it is possible to obtain a value of 2.8. From the DSP point of view, one can get a 2.78 gain factor. This is one indicator.

On one hand, one thing that is not integrated in such benches is the fact that a complete application merges many kinds of functions. This means that the optimization is less efficient for part of the implemented algorithm. Furthermore, the application integrates several function calls that add, in some cases, significant overhead.

On the other hand, such benches do not assume that the maximum potential of the DSPs is exploited. One preferably measures the gain factor to get effective comparison criteria for the go-no go decision because the developer should go further down in assembly optimization.

Thus, if the generic C implementation cycle count indicates more than five to six times the number of targeted cycles, the developer may consider that the real-time application is not reachable in a reasonable amount of time.

Stage G3: Reference C Code/Optimization

This implementation is the reference after generic code qualification to be optimized at the C level. Based on this code, one applies several rules concerning the method of implementing the application that fosters processing time reduction.

A primary objective is to establish a test process that guarantees the integrity of the processing. The goal is to reduce the cycle consumption and not to transform the result of the processing. It is also possible to establish a specific test script to validate the optimization and/or use tools to compare the processing results.

The script makes easier the run of several tests and allows the programmer to gather information (traces) on the application behavior. The tools allow the creation of specific comparisons on the processed data. The Cadence Cierto Signal Processing Worksystem (SPW) is capable of such a task and can speed the development cycle.

Stage G3.1: Optimization

For the high-level language implementation, optimization preferably uses tricks such as loop reduction, loop merging, test reduction, pointer usage, and in-line functions or macros, to reduce context switching. These tricks are generic and can be used for most if not all high-level languages.

Another optimization step that can be integrated at this level is development chain optimization by addressing the specific options of the pre-processor, compiler, assembler, and/or linker. This may be useful if the implementation is initially done with the target development environment. Generally, the applications are initially developed on PCs or Workstations. Then, taking advantage of the

generic compiler is not useful and can lead to bad decisions in terms of performances and code size.

At the implementation level, the developer assumes that the target DSP is fixed and that a simulator is available. Many C optimizations as are known in the art are possible at this language level.

Stage G3.2: Profiling

Each time a specific implementation is globally applied and validated the result is preferably benched, and if possible, fixed to facilitate further optimization. Preferably, at least three parameters are integrated: the global effort in terms of time to integrate a new optimization step, the processing time reduction that can be evaluated, and the code size evolution. These parameters preferably are correlated to the time dedicated to the project and whether or not the application is mandatory to system functionality.

Processing Time Reduction

In most cases, the gain in processing time follows a x^{-1} law. In-line functions, loop reduction and/or unrolling produce significant gain. Integrating pointers are normally less significant. However, a curve like that presented in FIG. 4, which regroups the measures realized for the generic implementation process, may be obtained.

A goal of these measures is to understand the impact of a modification. There is no generic rule that can be applied to all the code and all of the applications that reduce the number of cycles. Modifications that appear to optimize cycle count can actually increase it. Another goal is to fix a limit for the different optimization steps in terms of time. One rule, for example, may be to measure more than five percent of cycle reduction between two steps.

Code Size Evolution

This development measure is necessary to have embedded applications that do not have several Mbytes of memory available on the final system. Experiments have shown that the generic C optimization process considerably increases the code size. A fully dedicated C optimization process generally decreases the size. However, the programmer preferably guarantees that the code size does not exceed the available memory size of the target system.

Specific Implementation Process (Target-dependent Implementation in the High-level Language)

In this process, some instructions to allow the use of DSP-specific characteristics preferably are integrated into the C or other high-level language implementation. Many of the instructions may be addressed by using *pragma* instructions that are placed in the code to take advantage of caches or internal RAM, loop counters, multiply-accumulate capabilities (MAC), and multiply-subtract

capabilities (MSU). Other specific characteristics like *splitable* ALUs or multipliers, parallel instruction execution, and pipeline effects are addressed in the assembly level. For some DSPs, the only way to use these characteristics is to handle them at the assembly level. Furthermore, this step requires that the developer perform the least amount of tuning on the code to comply with the DSPs features.

Although the *pragmas* and intrinsics tend to detract from the portability, those parts of the code may be encapsulated and isolated. With the use of "#if-define" or other such conditional compiling flags, target compiler dependent flags can be integrated into the code so that it is possible to recompile the same application for all the targets to be addressed. However, this method of implementation requires a clear and structured versioning system as well as clear coding rules. One of the main issues arises from the need to support more than three or four different targets.

Another task of this stage is to implement the high-level language (e.g., C) code and look at the effect obtained on the generated assembler. The goal is not to modify assembly code but to write C code in a way that the assembler part of the compiler generates optimized assembly code. The assumption is made that there is some specific C implementations that will impact the generated assembly code in the same way for many compilers. The examples are the "do {} while" or the MAC integration. However, this is mainly true for the second and third DSP generations.

One can also use the example of the post-register modification. If the developer has realized a conversion of the implementation to integrate pointers, the position in the code of the pointer increment automatically generates or not the post-register modification in the assembly.

FIG. 5 is a flow diagram depicting a simplified representation of the main steps of the specific implementation process 500 represented as step 106 in FIG. 1. Detailed steps of the specific implementation process are depicted in the task flow diagram of FIG. 6.

Stage Sl: C-Optimized Code Impacting the Assembler

A key objective is to integrate specific *pragmas* and intrinsics into the code. The *pragmas* allow the use of cache or internal RAM memories and integration of loop counters to optimize loop branches. The other aspect of this optimization concerns the implementation modifications that take advantage of the specific capabilities of the target DSP, including multiply-accumulate, multiply-subtract, splittable multiply-add, and post register modification.

The goal is to generate the assembly code and observe what can be modified in the C implementation that can be translated differently by the compiler.

Stage S2: Specialized Low-Level Functions

Depending on the implementation structure, it may be necessary to tune some specific functions that are used intensively. Some methods of accomplishing this include, for example, removing code that is not used, avoiding overhead introduced by recursive calls, moving loop invariant expressions out of the loops, and reducing the scope of the variables (using macros integrates this concept naturally).

Code Portability

While the above-mentioned improvements may be viewed as non-portable, they are portable in the sense that the overall architecture of the implementation can be ported and reused. To achieve that, the developer preferably encapsulates the specific instructions of a DSP by integrating specific flags related to the target compiler in the code, and by using a source versioning system to handle the various target DSPs. Note that integrating specific flags can validate specific code parts depending on those flags.

FIG. 7 is a graph depicting examples of curves of the evolution of a software application with respect to its performance and size in a target-dependent optimization process. As shown in FIG. 7, the size decreases slowly because specific points of the application are addressed, but the impact on the performance can be impressive.

Fully-Dedicated Implementation Process

A fully dedicated implementation process is the lowest stage of the development process. Trade-offs on the application are preferably made by removing some processing passes, wherever possible. Assembly-specific optimization is also integrated to finally reach the target performance.

A key challenge is to determine after a profiling if the performance goal is reached. FIG. 8 is a task flow diagram depicting steps of a dedicated implementation process 800 represented as step 108 in FIG. 1. The dedicated implementation process includes two main steps, manual assembly optimization and feature tuning/cutting.

Manual Assembly Optimization

At this point, very low-level assembly language optimization is integrated. Key characteristics for this implementation generally come from parallel instructions, pipeline effects, and not fully optimized assembly code. Regarding parallel instructions, some DSPs are able to execute several instructions in the same clock cycle. It is possible to execute loads-operation-store in the same instruction. The main objective is to be able to integrate the pipeline effects that affect the availability of the processed data.

With respect to pipeline effects, mainly in the branch call is it possible to code specific instructions and take advantage of the pipeline delay slots. This

optimization can be useful for the loop intensive applications. It is mandatory to handle the parallel instruction optimization.

For the computationally intensive part of not fully optimized assembly code, it may be necessary to reorganize the generated code and integrate a more optimal way of using accumulators and registers.

Feature Tuning/Cutting

Depending on the capacities of the used DSP, it may be necessary to re-adapt the application because it does not fit. If such a decision is made, then the high-level steps of the process are not adapted or have been neglected. It is necessary to re-evaluate the application behavior in terms of processing, which is normally a high-level task of the process, for example, floating to fixed-point implementation.

One work-around is to drop out some specific part of the application that will have little impact on the quality of the processed data. For example, in an audio processing application, functions such as a ring subtraction, a high-pass filter on the input signal, or compression rate could be dropped without a significant loss of performance. Although the gain in terms of performance may not be high, cutting compression rate can suppress enough cycles to reach the target performance.

Reaching this step of the dedicated implementation process may mean that the application has not been evaluated correctly. If this is the case, then optionally, some of the highest process levels may be re-addressed.

While preferred embodiments of the invention have been described herein, and are further explained in the accompanying materials, many variations are possible which remain within the concept and scope of the invention. Such variations would become clear to one of ordinary skill in the art after inspection of the specification and the drawings. The invention therefore is not to be restricted except within the spirit and scope of any appended claims.